Comparing Optimisation Algorithms on the CIFAR-10 Dataset

1st James Prestwich
Msc Data Science
University of Surrey
Guildford, UK
jp01702@surrey.ac.uk
URN: 6772299

2nd Pauline Schouten
Msc Data Science
University of Surrey
London, UK
ps00711@surrey.ac.uk
URN: 6520325

3rd Khalil El Daou
Msc Data Science
University of Surrey
Guildford, UK
ke00338@surrey.ac.uk
URN: 6760445

***Abstract*—In image classification, selecting the appropriate optimisation algorithm to train the weights of a neural network is key to achieving good performance. This work explores different types of existing optimizers such as gradient descent based algorithms and evolutionary algorithms. Then, modifications of leading optimizer SGD were carried out in order to maximise accuracy on the CIFAR10 image dataset using a Convolutional Neural Network. A separate approach was tested where both the accuracy and the sum of the square weights were optimized as a bi-objective problem.**

***Index Terms*—Image classification, CNN, Evolutionary algorithms, SGD, CIFAR10**

I. Literature Review

Image classification involves the processing of pixel data leading to a large number of parameters for a neural network to update. The most widely used network architecture for image classification problems is the convolutional neural network (CNN) [1]. It is well-suited to image recognition due to a convolutional layer that reduces the high dimensionality of images without losing its information.

The training of CNNs can be highly optimized to reduce not only the loss but also the training time. There are 2 main approaches to optimizers in deep learning, the first is based on simple gradient descent.

Batch gradient descent [2] computes the gradients, at each training instance, which are then multiplied by a defined learning rate to update the weights in the network. This can be very slow as for each step it uses the whole training dataset for gradient calculation. This algorithm is therefore not suitable for very large datasets. It will often make unnecessary calculations of the gradient when there are similar training data points. This kind of vanilla gradient descent optimizer is rarely used in neural network training, especially in image classification where there are large training datasets. This would require too much memory for Batch Gradient Descent.

An improved version of this is Stochastic Gradient Descent [3]. Instead of using the whole training dataset for each step, it picks an instance of training data at random to compute the gradient. This is a faster method however SGD can easily get stuck in local minima or plateaus due to the fixed step size. The SGD algorithm is frequently used with very large training datasets due to how it generalizes very well with extensive training. It is therefore well-suited to image classification problems.

Finally, Mini-batch Gradient Descent [4] is an approach that computes the gradients for a subset of the training data which allows for fast convergence and adjusting for the noise. The larger the batch size, the less noise. However, the same issues that the SGD faces are present here. There is no guarantee of convergence due to getting stuck in local minima/maxima and the fixed learning rate must be chosen manually. This algorithm is sometimes preferred over SGD as some believe that picking 1 training datapoint is not random enough and that this batch method improves on it.

The issues of gradient descent algorithms have tried to be solved by more recent types of optimizer: Adaptive Optimizers. The main idea of this approach is to have an adaptive learning rate.

The Adagrad algorithm [5] adjusts the learning rate slightly for frequent features and makes larger changes to it for rarer features. The weakness of this approach is the high chance of ending up with a very small learning rate so the weights stop being updated, and the networks stops learning. This algorithm is mainly used in Natural Language Processing (NLP) to train transformer models [6]. NLP deals with sparse data (containing many 0s or low values), so using Adagrad ensures that these frequent '0' values do not strongly impact the learning rate.

RMSprop, an unpublished adaptive learning method, improves on this by storing past gradients during training. This solves the issue of vanishing learning rates. Again, this algorithm is often applied to NLP problems or other datasets that deal with sparse features. Both Adagrad and RMSprop are less common in image classification. The preferred choice found in many image recognition and classification problems is ADAM.

The Adam optimizer [7] combines this approach of past gradient storage with momentum. This means it estimates the first and second moments of gradient to adapt the learning rate for each weight of the neural network. Adam is the most popular optimization algorithm in deep learning [8], it is well-known for achieving good performance with minimal hyper-parameter tuning.

Some works have implemented genetic algorithms (GAs) as an optimizer to train the weights [9] [10] or even to learn the best CNN structure [11]. As an optimizer, it avoids getting stuck in local minima/maxima through mutations. This is an improvement on SGD, however GAs are rarely the preferred choice due to extensive execution time compared to gradient descent methods [12].

Summary

Adaptive gradient methods provide quick results and require no extensive parameter tuning. Of all the adaptive methods, Adam is seen as the best choice for most cases. However recent research [28] suggests that even though Adam has faster convergence, SGD generalizes better than Adam and thus results in an improved final performance. Population-based optimizers are much less common but still can demonstrate good results as long as execution time is not a constraint.

Our Contribution

To achieve competitive performance on the CIFAR10 dataset, modifications will be made to the SGD algorithm. The aim is to get faster **convergence** and higher final **testing accuracy** than ADAM, the leading training algorithm. Learning rate, Momentum, Dampening, Weight Decay and Nesterov Momentum are all hyper-parameters that can be experimented with to achieve better performance.

II. Neural Network Architecture

As discussed in the literature review, CNNs are the preferred class of network in image recognition. For classification of the CIFAR10 images, various leading CNN models such as ResNet101 [13] and LeNet [14] were tested. Although the ResNet101 model achieves high accuracy on the dataset, the extensive training times meant that they were not feasible for later implementation with Generational Algorithms. LeNet achieves a low accuracy on the CIFAR10 dataset. Therefore, a simpler convolutional neural network was built based on online resources [15]. The basic architecture, shown in Fig.1, is 6 convolutional layers with 3 pooling layers and then followed by 3 fully connected layers.

Using more convolutional layers leads to better accuracy however the training time quickly increases and so does the risk of overfitting. Experimenting with more than 6 led to excessive training times. Less than 6 majorly reduces the testing accuracy.

Setting default kernel size, padding and stride for each convolutional layer:

1) Kernel size is 3 (smaller kernel leads to overfitting, larger leads to underfitting, 3 is a common choice as a trade-off)
2) Padding is 1 (ensures input image size and output feature size is the same)
3) Stride is 1 (standard convolution, no pixel skipping)

The **ReLU activation** function that gets called after each layer increases the non-linearity in the images which allows

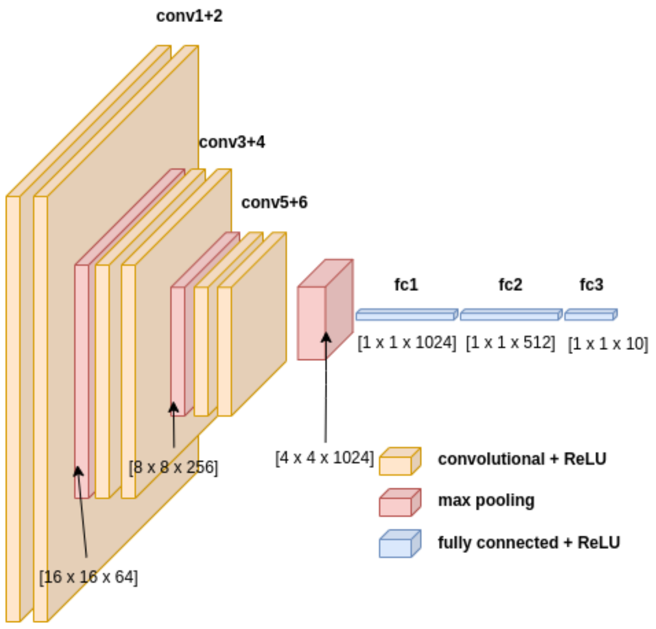
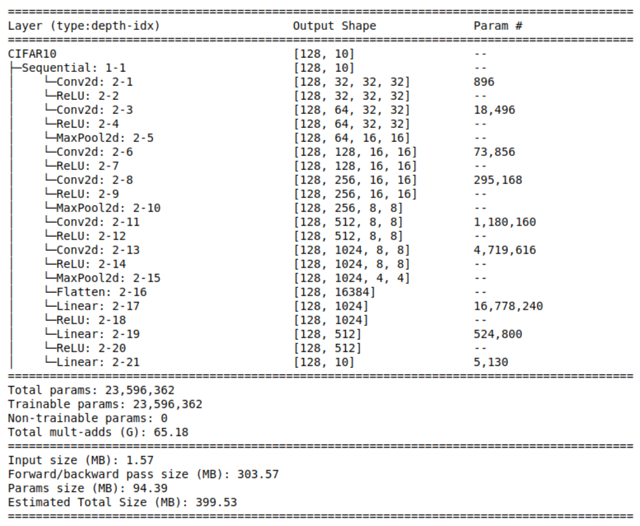


Fig. 1. Diagram showing the detailed structure of the selected CNN architecture.



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
CIFAR10                                  [128, 10]                 --
├─Sequential: 1-1                        [128, 10]                 --
│    └─Conv2d: 2-1                       [128, 32, 32, 32]         896
│    └─ReLU: 2-2                         [128, 32, 32, 32]         --
│    └─Conv2d: 2-3                       [128, 64, 32, 32]         18,496
│    └─ReLU: 2-4                         [128, 64, 32, 32]         --
│    └─MaxPool2d: 2-5                    [128, 64, 16, 16]         --
│    └─Conv2d: 2-6                       [128, 128, 16, 16]        73,856
│    └─ReLU: 2-7                         [128, 128, 16, 16]        --
│    └─Conv2d: 2-8                       [128, 256, 16, 16]        295,168
│    └─ReLU: 2-9                         [128, 256, 16, 16]        --
│    └─MaxPool2d: 2-10                   [128, 256, 8, 8]          --
│    └─Conv2d: 2-11                      [128, 512, 8, 8]          1,180,160
│    └─ReLU: 2-12                        [128, 512, 8, 8]          --
│    └─Conv2d: 2-13                      [128, 1024, 8, 8]         4,719,616
│    └─ReLU: 2-14                        [128, 1024, 8, 8]         --
│    └─MaxPool2d: 2-15                   [128, 1024, 4, 4]         --
│    └─Flatten: 2-16                     [128, 16384]              --
│    └─Linear: 2-17                      [128, 1024]               16,778,240
│    └─ReLU: 2-18                        [128, 1024]               --
│    └─Linear: 2-19                      [128, 512]                524,800
│    └─ReLU: 2-20                        [128, 512]                --
│    └─Linear: 2-21                      [128, 10]                 5,130
==========================================================================================
Total params: 23,596,362
Trainable params: 23,596,362
Non-trainable params: 0
Total mult-adds (G): 65.18
==========================================================================================
Input size (MB): 1.57
Forward/backward pass size (MB): 303.57
Params size (MB): 94.39
Estimated Total Size (MB): 399.53
==========================================================================================
```

Fig. 2. Breakdown of each layer in the selected CNN architecture.

for a more powerful model. The **max pooling** layers reduce the output size, there is one after every 2 convolutional layers. 2 by 2 max pool was selected. This down-sampling method helps keep the dimensionality of the data low and reduces the effect of image invariances (rotated images, shifted images or scaled images). The **fully connected** layers take the output of pooling and assign the best label for each image, thus the final output size of fc3 is the number of classes in the CIFAR10 dataset: 10.

III. Training Algorithm

After loading the data and dividing it into training and testing data, the chosne neural netowrks was trained and evaluated on CIFAR10's testing data. To train the model, an optimizer was needed. There are various optimizers to choose from, but for our model Adam and SGD optimizers were used. Adam shows better accuracy at first and converges faster than SGD, but SGD performs and generalizes better at the end [16]. Gradient Descent is the most used optimizer for training neural networks. The optimizer updates the weights of the model and study the variations as the objective functions reaches a minimum [16]. SGD is a form of Gradient Descent that work on a small random subset of the dataset to avoid redundancies [16]. Adam is an adaptive optimizer that calculates and changes the learning rates of the parameters as the model is training [16]. The objective of this study is to finetune the SGD optimizer for it to converge faster than ADAM and stay consistent and precise at the end of the training.

Before tuning parameters, both optimizers were tested in their default state.

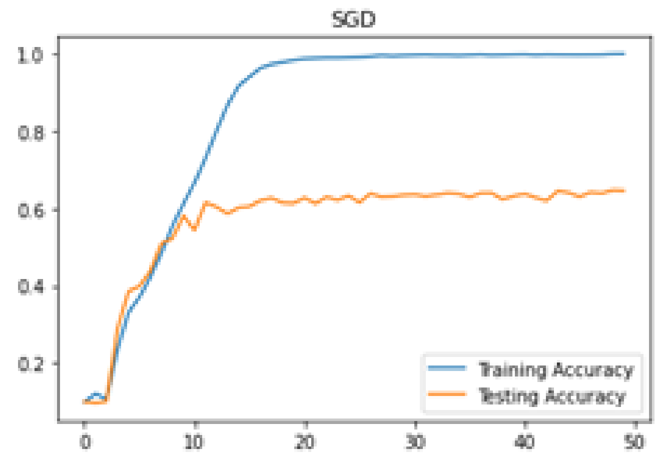


Fig. 3. Plot of default SGD with learning rate of 0.1.

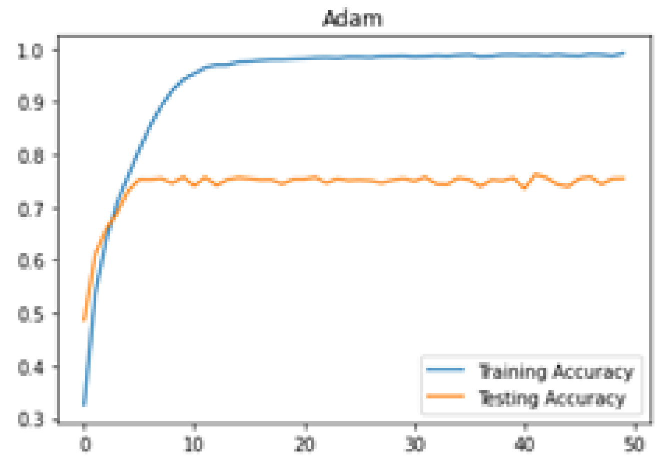


Fig. 4. Plot of Adam Optimizer with learning rate of 0.0003.

After training on 50 epochs, it was found that ADAM was converging a lot faster than SGD. It was taking ADAM around 4 epochs to converge to a training accuracy of around 75%, whilst SGD was taking around 8 epochs to converge to a training accuracy of around 62%.

Fine Tuning SGD Hyperparameters:

To improve SGD, different hyperparameters were assessed. To assess different values for the hyperparameters, an algorithm was created to study different combinations of hyperparameters, train different models, and get the best hyperparameters that can be utilized later while training. The hyperparameters for SGD that were assessed on are learning rate, momentum, dampening, weight decay, and Nesterov momentum. Below are the definitions of the SGD hyperparameters:

1) **Learning Rate**: It is a parameter that shows how fast the function should move as its searching for the local minima or the solution. (Default: 0.01) [17].
2) **Momentum**: It is used to help the function converge faster and improve the optimization process. It ranges from 0 to 1. (Default: 0) [18].
3) **Dampening**: It ensures the optimizer' loss is not jumping and taking bigger steps. It ranges from 0 to 1. (Default:0) [19].
4) **Weight Decay**: It adds a penalty to the weights (L2 norm) of the optimizer to decay the weights and normalise the loss function (regularization function). It ranges from 0 to 1. (Default: 0) [20].
5) **Nesterov Momentum**: It is an updated version of the momentum mentioned before that uses the partial derivative of the weight updates. It is a Boolean value. (Default: False) [21].

A flowchart of the training algorithm is shown in Fig. 5.

After running the algorithm at 20 epochs, the best parameters to use were: [learning rate, momentum, dampening, weight decay]:

1) (0.01, 0, 0, 0, False) - 40% and still going (default sgd with lr=0.01)
2) (0.01, 0, 0.5, 0, False) - 28% and still going.
3) (0.01, 0, 1, 0, False) - 38% and still going
4) (0.01, 0.5, 0, 0, False) - 52% and still going
5) (0.1, 0, 0, 0, False) - 60% testing (default sgd with lr=0.1)
6) (0.1, 0, 0.5, 0, False) - 65% testing
7) (0.1, 0, 1, 0, False) - 65% testing
8) (0.1, 0.5, 0, 0, False) - 75% testing
9) (0.1, 0.5, 0.5, 0, False) - 68% testing

With a learning rate of 0.01, it was observed that the model was using all 20 epochs and still has not converged. The best one for a learning rate of 0.01 was with a momentum of 0.5 that showed 52% testing accuracy and still going. Fig. 6 shows a plot for the best optimizer with a learning rate of 0.01 and a momentum of 0.5.

For a learning rate of 0.1, the model was converging at 5-7 epochs and showing better accuracy than 0.01. The best one with a learning rate of 0.1 was with a momentum of

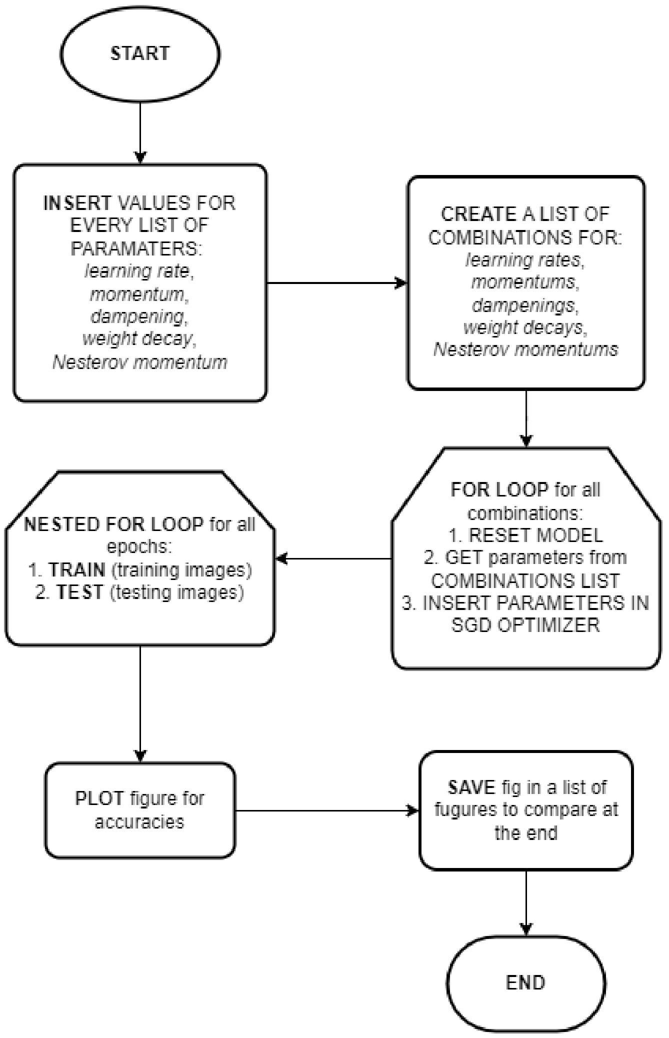

START
INSERT VALUES FOR EVERY LIST OF PARAMATERS: learning rate, momentum, dampening, weight decay, Nesterov momentum
CREATE A LIST OF COMBINATIONS FOR: learning rates, momentums, dampenings, weight decays, Nesterov momentums
FOR LOOP for all combinations: 1. RESET MODEL 2. GET parameters from COMBINATIONS LIST 3. INSERT PARAMETERS IN SGD OPTIMIZER
NESTED FOR LOOP for all epochs: 1. TRAIN (training images) 2. TEST (testing images)
PLOT figure for accuracies
SAVE fig in a list of fugures to compare at the end
END

Fig. 5. Flowchart for fine-tuning hyper-parameters of SGD.

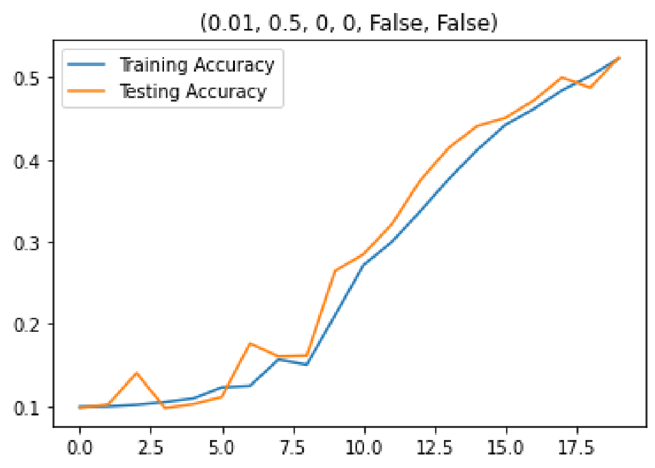


Fig. 6. A plot of SGD with lr=0.01 and momentum=0.5

0.5 only that showed a 75% training accuracy and converging at 5 epochs. The second best was using momentum of 0.5 and dampening of 0.5 that showed 68% training accuracy and converging at 5 epochs also.

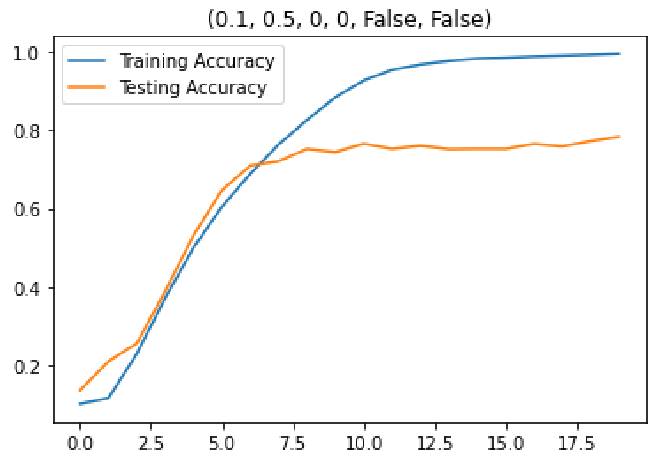


Fig. 7. A plot of SGD with lr=0.1 and momentum=0.5

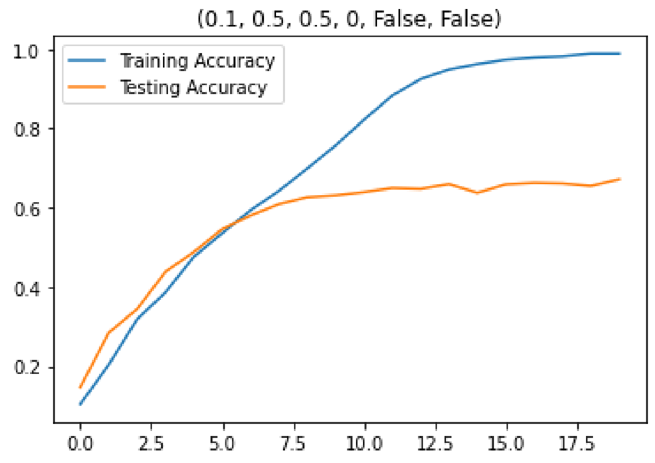


Fig. 8. A plot of SGD with lr=0.1, momentum=0.5, and dampening=0.5

Using weight decay was underfitting most of the time. That is because high values for weight decays were tested on. So, in the adaptive SGD, explained in the next section, a small amount of decay was added to the SGD optimizer.

Adaptive SGD:

Another way to improve SGD is by trying to change the learning rate as the model is training. Adaptive SGD will show better results as the learning decreases over time. For our algorithm, after knowing what hyperparameters to use, a step decay and an exponential decay were examined on the learning rate of the SGD optimizer. A step decay will reduce the learning rate by a gamma value every set of epochs [23]. Exponential decay will reduce the learning rate exponentially [24]. For both decays to be achieved, a scheduler is used in the code to step and change the learning rate while in the training loop.

A flowchart for adaptive SGD is shown Fig. 9.

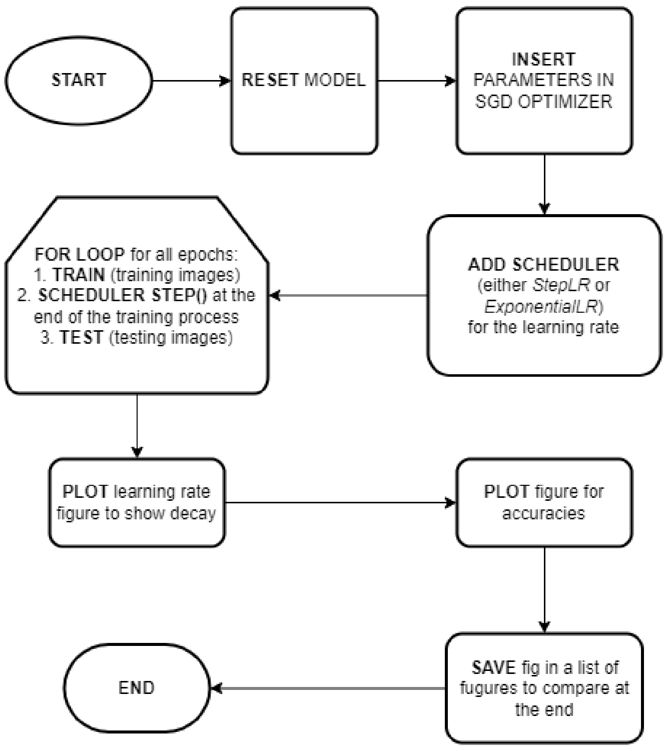

START
RESET MODEL
INSERT PARAMETERS IN SGD OPTIMIZER
ADD SCHEDULER (either StepLR or ExponentialLR) for the learning rate
FOR LOOP for all epochs: 1. TRAIN (training images) 2. SCHEDULER STEP() at the end of the training process 3. TEST (testing images)
PLOT learning rate figure to show decay
PLOT figure for accuracies
SAVE fig in a list of fugures to compare at the end
END

Fig. 9. Flowchart for adaptive decay for learning rate of SGD.

Exponential and Step decay were tested on, because these two techniques are the most common decay techniques used for adapting and changing the learning rates for SGD optimizer [27].

A plot of how the step decay is changing the learning rate for every 2 epochs as the model is training is shown in the Figure 11, and a plot of how the exponential decay is changing the learning rate as the model is training is shown in the Figure 11.

After running the adaptive SGD algorithm and tuning the rates of the decay of the learning rate, exponential decay showed better results than step decay as seen in the Figure 12.

IV. Results

In order to directly compare ADAM, SGD and GA as optimizers in the neural network, first all layers apart from the final layer were optimized using ADAM. This was because there were over 23 million weights in the network but only 5130 in the final layer. When there are this many parameters, the computational cost becomes prohibitively high for GAs, they also performed very poorly when used for this in preliminary testing. This is because when the genes involved become too complex, they take a very long time to improve [25]. After this each of the three optimizers were used on just the final layer.

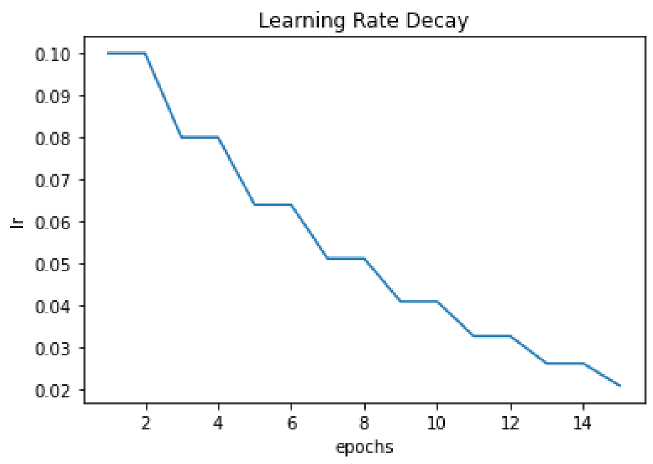


Fig. 10. A plot showing the step learning rate decay.

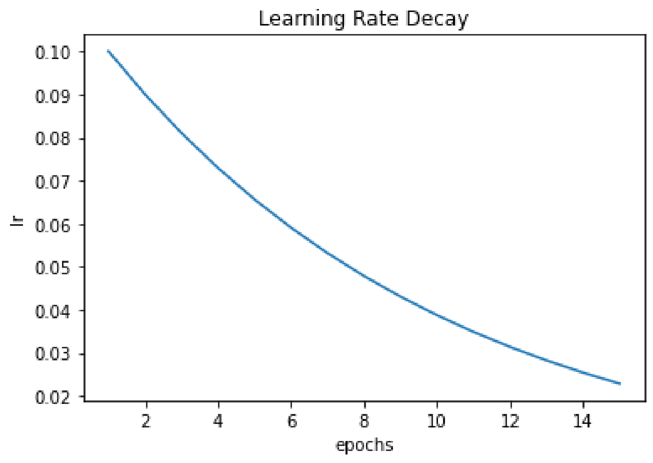


Fig. 11. A plot showing the exponential learning rate decay.

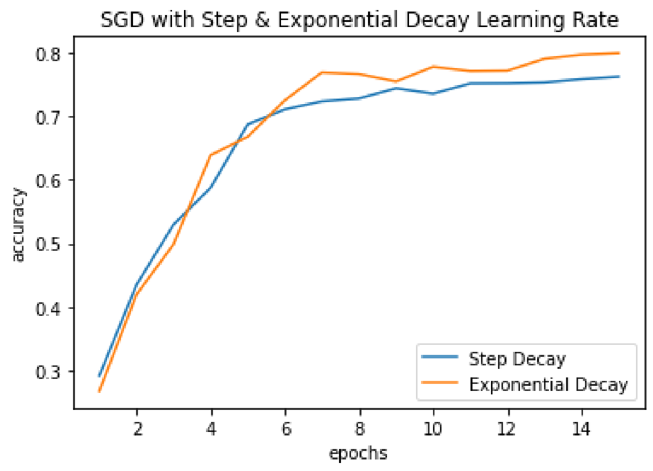


Fig. 12. A plot showing the the comparison between step and exponential learning rate decay.

The initial layers were trained using ADAM as this is the main comparative algorithm. This training was done over nine epochs through the training data with an image batch size of 128. Nine epochs were chosen as this is the point at which the testing accuracy stopped increasing and the model started overfitting the data. Each of the three optimisation algorithms were then iterated 200 times with a batch size of 500 images from the training data, for ADAM and SGD this corresponds to a forward and backward pass for GA it corresponds to one generation of the population. Further details on the parameters used the GA can be found in table I.

TABLE I
PARAMETERS USED IN THE GENETIC ALGORITHM

Parameter	Value
Population Size	100
Number of bits to represent each weight	10
Number of parameters	5120 layer weights and 10 bias weights
Crossover probability	0.6
Type of crossover	Two-point crossover
Bit mutation probability	1 / (gene size) or 1/(number of weights * number of bits representing each weight)
Gene mutation probability	0.15
Batch size of images	500
Number of generations	200

Fig. 13 shows how each of the algorithms performed on the test data per iteration. It can be seen that GA performed poorly, not only did it take longer to optimize it was also unable to achieve an accuracy greater than 47%. ADAM and SGD both performed well, managing to achieve an accuracy of 80% but the SGD with tuned hyperparameters was faster at optimising the network and achieved marginally better maximum and final testing accuracies as shown in table II.

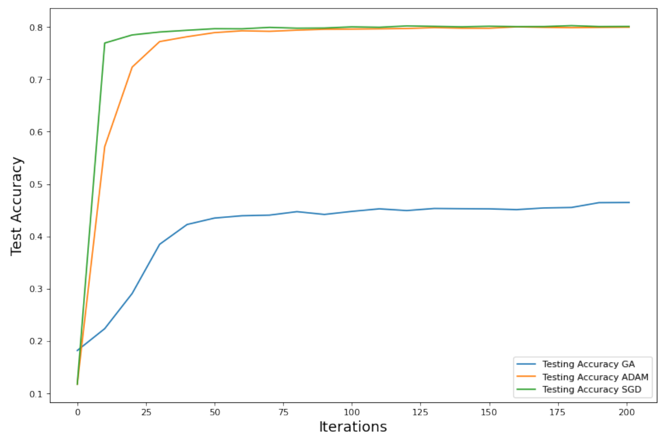


Fig. 13. A graph comparing the test accuracy every ten iterations for each of the three optimizers when training the final layer of the neural network.

A comparison between the best tuned parameters is plotted in the Figure 14 after training on the whole model layers.

It is obvious that using a learning rate of 0.01 for the whole model will take a long time to converge as seen in the figure above. After 15 epochs, the SGD optimizer with a learning rate of 0.01 reached around 55% and it still hasn't converged. So, for our model, an increase in the learning rate to 0.1 will suffice better results to compare to the Adam optimizer. Adam was the fastest one in converging after only 6 epochs. However, the exponential decay SGD showed better accuracy at end of the training and have never dropped down across all epochs. An SGD with a 0.1 learning rate, 0.5 momentum, and 0.0001 weight decay was outperforming all SGD optimizers except at the end when the exponential decay showed better accuracy after epoch 13. However, the SGD optimizer with the weight decay showed some drops in the middle of training in contrast with exponential decay. The step decay was outperformed by most optimizers, so a change in step size (number of epochs to change learning) could be tuned on a bit more.

TABLE II
COMPARISON OF ACCURACIES ON FINAL LAYER OF NEURAL NETWORK

Optimisation Algorithm	GA	ADAM	SGD
Max Testing Accuracy (%)	46.46	80.01	80.24
Final Testing Accuracy (%)	46.46	70.94	80.08

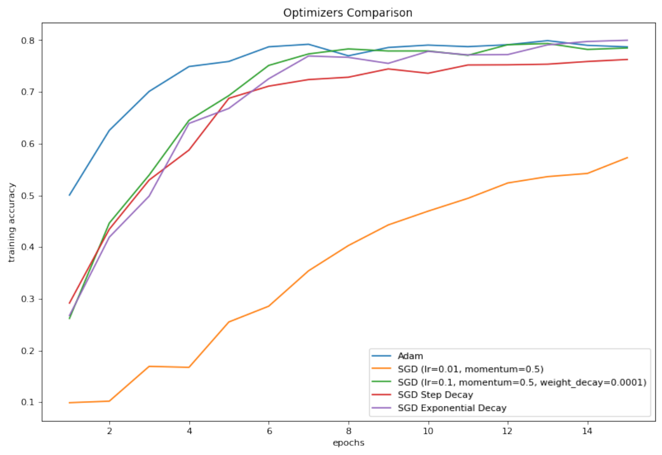


Fig. 14. A plot showing the the comparison of all optimizers training the whole model

From the results, GAs are unsuitable for optimising large convolutional neural networks such as the one used in this report to classify the CIFAR-10 image dataset. Not only do they have a far greater computational cost than gradient descent based algorithms, they also do not achieve the same accuracy. Hitano [25] writes: "Although genetic algorithms converge efficiently at the initial stage, the speed of convergence is severely undermined later due to its weak local fine-tuning capability." This was evident in this report as the GA would get stuck in local minima after initial improvement. This is likely because population changes did not produce significant improvement in the cross-entropy loss nor were improvements consistent as the image batch changed. This effect could have been mitigated by using a larger image batch size, but this was limited by the amount of GPU storage available.

The results showed that contrary to the research discussed in the literature review, SGD can be made to converge faster than ADAM by adjustment of the hyper-parameters, especially using momentum. For the neural network in this report, with a momentum of 0.5 it converged faster when optimising the final layer as well and it did this at no extra computational cost. However, it converged slower when optimising the complete neural network compared to ADAM. SGD was also able to achieve a marginally better accuracy on the test data in both cases.

V. Training as a Multi-objective Optimisation

Next, the final layer of the neural network was trained as a bi-objective problem where the objectives were accuracy on the images as well as minimisation of the sum of the square of the weights (Gaussian regularization). This is because as training goes on typically the size of the weights increase as the data is fit better and better to the training dataset; larger weights indicate the model is growing too complex and is overfitting the dataset. Using non-dominated sorting genetic algorithm (NSGAII) to encourage selection of smaller weights should create a model that generalises better and therefore will perform better on the test dataset [26].

As per the previous optimisation by GA, only the final layer of the network was trained due to the large number of parameters. For each batch of training images, the cross-entropy loss was calculated along with the sum of the squared weights in the final layer to be used as the fitness for each individual in the population. For each generation there first was a selection tournament based on dominance followed by crowding distance to determine offspring for crossover and mutation. After this NSGAII was used to select the best individuals from both the original population and the offspring which would continue into the next generation. Further details on the parameter setup for this algorithm can be seen in table III.

TABLE III
Parameters used in bi-objective optimisation

Parameter	Value
Population Size	100
Number of bits to represent each weight	10
Number of parameters	5120 layer weights and 10 bias weights
Crossover probability	0.9
Type of crossover	Two-point crossover
Bit mutation probability	1 / (gene size) or 1/(number of weights * number of bits representing each weight)
Gene mutation probability	0.15
Batch size of images	500
Number of generations	200

As can be seen in Fig. 15 the cross-entropy loss stops decreasing significantly after 50 generations; this is very similar to the results found when using GA in the previous section. However, in Fig. 16 it can be seen that the sum of squared weights continues to decrease, this would indicate that the model may perform better on the test data, however, in Fig. 17 it can be seen there is a positive correlation between the sum of the squared weights and test accuracy. This shows that models with smaller weights not only performed worse on the training data they also performed worse on the test data.

However, when the final testing accuracy was compared to the final testing accuracy for the GA, NSGAII's final accuracy was significantly better at 59.59% compared to 46.46%. The bi-objective NSGAII also outperformed the GA on all testing runs as parameters were tuned. As can be seen in Fig. 16, the model was able to reduce the sum of the squared weights for the entire population without decreasing the cross-entropy loss. Even though the best solutions had greater weight values within the population, these weight values were dropping as generations went on, producing solutions that generalised better and outperformed previous generations on the test data. This shows that motivating the model to reduce the size of the weights is beneficial in the case of GAs. However, there is a caveat; the best accuracy achieved by this method was 60% compared to the 80% achieved by the gradient descent methods. Also, as can be seen in Fig. 16, the diversity in the sum of the squared weights was decreasing and seemingly converging, indicating that further improvement through future generations was likely to be small.

As earlier discussed, because GAs are unsuited for optimising large convolutional neural networks, NSGAII never managed to get close to overfitting the data. Therefore, it is difficult to tell whether the Gaussian regularizer conclusively helped produce models that generalised better. Further work is required to find out whether it is worthwhile training large CNNs as a multi-objective problem.

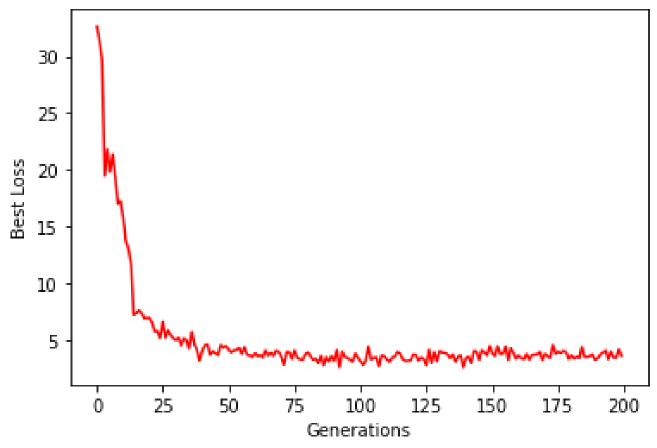


Fig. 15. A graph showing the cross-entropy loss of the best solution in each generation of the NSGAII model on the testing data

Conclusions

The main finding is that, when only training the final layer, the modified SGD optimizer performs best compared to ADAM and the genetic algorithm. It has faster convergence and a slight improvement on accuracy. The drawback is that,

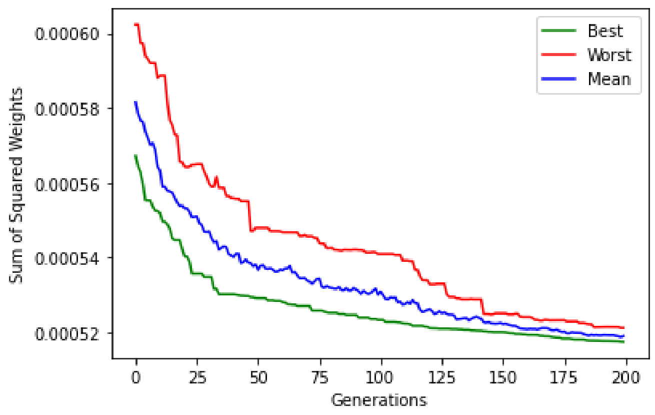


Fig. 16. A graph showing the cross-entropy loss of the best solution in each generation of the NSGAII model on the testing data

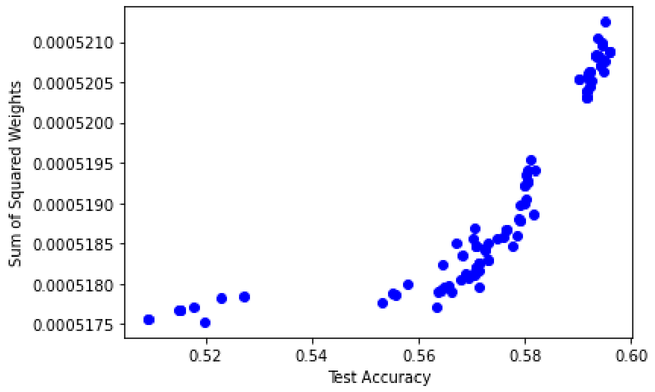


Fig. 17. A scatter plot of the optimal front from NSGAII against the two objectives, sum of the squared weights and accuracy on the test data.

when training with each of these methods on the entire model (all layers), ADAM surpasses the redesigned SGD optimizer both in convergence time and final accuracy. Moreover, it is evident that the genetic algorithm performs poorly in both cases, this is perhaps due to the large size of the neural network which does not suit population-based methods. However, the NSGAII results indicate that these methods may be better suited to a bi-objective approach where the sum of the square weights is also optimized.

In future, more complex modifications could be made to SGD such as or perhaps a hybrid approach of gradient descent and population-based methods could be trialled. Finally, given the promising performance of the bi-objective approach, this could be repeated with a gradient descent-based method instead of NSGAII.

References

[1] Neha Sharma, Vibhor Jain, Anju Mishra, An Analysis Of Convolutional Neural Networks For Image Classification, Procedia Computer Science, Volume 132, 2018, Pages 377-384, ISSN 1877-0509, https://doi.org/10.1016/j.procs.2018.05.198

[2] Lemaréchal, C. (2012). "Cauchy and the Gradient Method" (PDF). Doc Math Extra: 251–254. https://www.math.uni-bielefeld.de/documenta/vol-ismp/40_lemarechal-claude.pdf

[3] Bharath, B., Borkar, V.S. Stochastic approximation algorithms: Overview and recent trends. Sadhana 24, 425–452 (1999). https://doi.org/10.1007/BF02823149

[4] H. Wang, H. Luo, and Y. Wang, "Mbgdt: Robust mini-batch gradient descent," 2022. [Online]. https://arxiv.org/abs/2206.07139

[5] Duchi, J. C.; Hazan, E. & Singer, Y. (2011), 'Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.', J. Mach. Learn. Res. 12 , 2121–2159. https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

[6] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. In Advances in neural information processing systems (pp. 5998-6008).

[7] Kingma, D. P. & Ba, J. (2014), 'Adam: A Method for Stochastic Optimization' , cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. https://arxiv.org/pdf/1412.6980.pdf

[8] Karpathy, Andrej, "A Peek at Trends in Machine Learning", Medium, 2017 https://karpathy.medium.com/a-peek-at-trends-in-machine-learning-ab8a1085a106

[9] X. Yao. Evolving Artificial Neural Networks. Proceedings of the IEEE, 87(9):1423–1447, 1999.

[10] S. Ding, H. Li, C. Su, J. Yu, and F. Jin. Evolutionary Artificial Neural Networks: A Review. Artificial Intelligence Review, 39(3):251–260, 2013.

[11] Xie, Lingxi and Alan Loddon Yuille. "Genetic CNN." 2017 IEEE International Conference on Computer Vision (ICCV) (2017): 1388-1397.

[12] Stastny, Jiri & Skorpil, Vladislav. (2007). Genetic algorithm and neural network.

[13] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90.

[14] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.

[15] Asif, A. (2022) Training a convolutional neural network (CNN) on CIFAR-10 Dataset, Medium. students x students. Available at: https://studentsxstudents.com/training-a-convolutional-neural-network-cnn-on-cifar-10-dataset-cde439b67bf3 (Accessed: December 6, 2022).

[16] T. A. Team, "Why Should Adam Optimizer Not Be the Default Learning Algorithm? – Towards AI." https://towardsai.net/p/l/why-should-adam-optimizer-not-be-the-default-learning-algorithm (Accessed Dec. 07, 2022).

[17] A. Rakhecha, "Understanding Learning Rate," Medium, Jul. 07, 2019. https://towardsdatascience.com/https-medium-com-dashingaditya-rakhecha-understanding-learning-rate-dd5da26bb6de

[18] J. Brownlee, "Gradient Descent With Momentum from Scratch," MachineLearningMastery.com, Feb. 04, 2021. https://machinelearningmastery.com/gradient-descent-with-momentum-from-scratch (Accessed Dec. 07, 2022).

[19] "Hasty.ai," Hasty.ai. https://hasty.ai/docs/mp-wiki/solvers-optimizers/momentum-sgd/dampening-sgd (accessed Dec. 07, 2022).

[20] S. Yang, "Deep learning basics — weight decay," Analytics Vidhya, Sep. 05, 2020. https://medium.com/analytics-vidhya/deep-learning-basics-weight-decay-3c68eb4344e9

[21] J. Brownlee, "Gradient Descent With Nesterov Momentum From Scratch," MachineLearningMastery.com, Mar. 16, 2021. https://machinelearningmastery.com/gradient-descent-with-nesterov-momentum-from-scratch (Accessed Dec. 07, 2022).

[22] "SGD — PyTorch 1.13 documentation," pytorch.org. https://pytorch.org/docs/stable/generated/torch.optim.SGD.html

[23] "StepLR — PyTorch 1.13 documentation," pytorch.org. https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.StepLR.html#torch.optim.lr_scheduler.StepLR (Accessed Dec. 07, 2022).

[24] "ExponentialLR — PyTorch 1.13 documentation," pytorch.org. https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ExponentialLR.html#torch.optim.lr_scheduler.ExponentialLR (Accessed Dec. 07, 2022).

[25] Kitano, H. (1990, July). Empirical studies on the speed of convergence of neural network training using

[26] Peter L. Bartlett. 1996. For valid generalization, the size of the weights is more important than the size of the network. In Proceedings of the

9th International Conference on Neural Information Processing Systems (NIPS'96). MIT Press, Cambridge, MA, USA, 134–140.

[27] Suki Lau, "Learning Rate Schedules and Adaptive Learning Rate Methods for Deep Learning," Medium, Jul. 29, 2017. https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1

[28] Pan Zhou, Jiashi Feng, Chao Ma, Caiming Xiong, Steven Hoi, and E. Weinan. 2020. "Towards theoretically understanding why SGD generalizes better than ADAM in deep learning". In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20). Curran Associates Inc., Red Hook, NY, USA, Article 1787, 21285–21296. https://arxiv.org/abs/2010.05627

Appendix

Individual Contributions

James Prestwich:

- Researched and tested some convolutional neural networks.
- Adapted the neural network structure to allow for layers to be trained in isolation.
- Created and implemented the GA for optimising the final layer.
- Wrote the code to allow for fair comparisons of GA, SGD and ADAM on the final layer.
- Created and implemented NSGAII for optimising the final layer as a multi-objective problem.
- Wrote the results section of the report as well as the initial formatting in Latex.

Pauline Schouten:

- Conducted the literature review of current network architectures and training algorithms
- Experimented with convolutional neural network architectures and selected most appropriate model
- Write-up of Abstract, Literature Review, Neural Network Architecture and Conclusions in the report

Khalil El Daou:

- Tuned hyperparameters for SGD
- Added an adaptive learning rate for SGD (step and exponential)
- Compared SGD optimizers and ADAM when training the entire neural network
- Wrote training algorithm section of report